Chapter-V

Optimizing Frequent Itemset Mining Algorithm using Compaction and Parallel Processing

5.1 Introduction

Frequent Itemset mining is a widely used technique in data mining. It is the base of many data mining techniques including association rules, clustering and classification. However, it suffers with performance bottlenecks when data size is extremely large. In this work, frequent item set mining is made efficient using parallel processing on Graphical Processing Unit. The advent of parallel GPU machines created an opportunity to achieve improved performance by reducing communication overhead. Here, multiple processing units CPU and GPU lie on the same machine and the communication between them is using memory bus. GPU has independent memory and processing units and it can be attached with CPU to improve computational speed. A parallel version of frequent item set mining algorithm is proposed. Mining is applied on vertically compact data which saves repeated scans of whole database. Overall performance is also improved since mining for different support values can be calculated using bitwise operations on the vertical data structure. ~3X - ~7X speed up was observed on GPU as compared to multi core CPU.

5.2 Motivation

The main challenges in data mining include huge data size, extending reusability and incremental data mining. The data volumes are exponentially growing, therefore time and cost efficient mining is an open challenge. Apriori and the FP-growth are often used to mine frequent patterns from a set of transactions. Both have limitations in terms of time and space. FP-tree growth gives best performance condition to the size of the tree being accommodated in primary memory. Other algorithms suffer from large time consumption due to repeated
scanning of the database or file for counting frequent item sets. There is a need
to develop better FIM mining methods.

5.3 Proposed solution

FIM algorithm is generally applied on boolean dataset where the value is set to 1 only if that item is present in the transaction else remains 0. Horizontally formatted data is first transformed into vertical format by scanning the data set once and then frequent item sets are found through boolean operations. To make the support count step faster, GPU is used as a supporting device. CUDA language, an extension of C is used for programming GPUs. The proposed method substantially reduces the computational costs. Overall methodology can be observed in Figure 5.1.

Figure 5.1 Process Flow
FIM Process Flow:

i. Read file
ii. Convert data into compact vertical format
iii. Transfer the data to GPU
iv. Count frequent itemsets using bit operations on GPU
v. Generate candidate sets and repeat steps iv and v till no more candidates are found
vi. Send output to CPU

5.4 System Requirements for CUDA

CUDA capable GPU
Operating System: Windows 7 XP, Vista, 7 or 8 or Windows server 2003 or 2008
Nvidia CUDA Toolkit
Microsoft Visual Studio 2010

5.5 Experimental Setup

Processor: - Intel(R) Core(TM) 2 Duo CPU E7500 @ 2.93GHz.
RAM: - 2.00 GB.
GPU: - NVIDIA GeForce GT 610 (CUDA Cores 48).
Operating System: Windows 7

5.6 About GPU

GPU is a co-processor and acts as a hardware accelerator for many graphics and non graphic processes. With GPUs, parallel computing has become more significant. Several processes can run concurrently on GPU. GPGPU is a technique for high-performance computing that uses graphics processing units.

In this work, GeForce GT 610 is used for experimentation. Steps for GPU installation are specified next:

- Gently fix the GPU on a PCI slot of the PC.
- Connect the power cables.
• Install the driver.
• Restart the PC to work on GPU.

5.7 Dataset Used:

The dataset chosen is T40I10D100K which is downloaded from FIMI repository [112]. The features of the dataset are mentioned below:

<table>
<thead>
<tr>
<th>#Item</th>
<th>Avg. Length</th>
<th>#Transactions</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>40</td>
<td>100,000</td>
<td>4%</td>
</tr>
</tbody>
</table>

Figure 5.2 GPU Image
Figure 5.3 Input Data Set

Input Data Format:

Record 1: 36 69 115 226 278 343 345 358 368 370 401 410 450 489 494 573 577 581 583 610 682 692 705 722 832 862 886 908 923 932 960 977

Record 2: 8 51 55 73 78 117 140 175 187 229 266 295 304 366 381 413 424 429 501 512 523 529 538 572 575 576 593 675 676 688 735 758 785 797 812 823 826 843 854 868 871 888 893 956 982

Record 1 shows the presence of item no. 36, 69, 115 .....977 respectively.

5.8 Experiment

In this experiment, the proposed approach scans the database once and converts input data into compact format by mapping every element of matrix to every bit of a 32 bit integer. The number of records in original file is 1 lac. and the
number of itemsets is 1000. Horizontal representation of data indicates transactions on y axis and itemsets on x-axis. Whereas, for vertical database representation, item sets are indicated on y-axis and transactions on x-axis. Therefore, at max a compact matrix of 1000*3125 is required to represent 1 lac record and 1000 items. Here, 3125 integers can represent 1 lac records by utilizing its individual bits for indicating boolean values. For example, if item 4 & item 6 are present then 4th & 6th bit of first integer in first row will be set to 1.

```
0 0 0 1 0 1 0 0 0 0 . . . 0
```

Individual values are read and for setting the bits bitwise OR operation is applied.

Let \( M[ ] = \{
2147483648, 1073741824, 536870912,
268435456, 134217728, 67108864, 33554432,
16777216, 8388608, 4194304, 2097152,
1048576, 524288, 262144, 131072,
65536, 32768, 16384, 8192,
4096, 2048, 1024, 512,
256, 128, 64, 32,
16, 8, 4, 2, 1
\}; //every value with one bit on at different positions

5.9 Horizontal Compaction

The format of horizontally compact data is like:
Table 5.1 Initial data format converted to Data Format in horizontal compaction

<table>
<thead>
<tr>
<th>Trans. No.</th>
<th>ItemNo.</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction No.</th>
<th>ItemNo #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integer 1:- 32 bits=32 items</td>
</tr>
<tr>
<td></td>
<td>Integer 2:- 32 bits=32 items</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>&quot;</td>
</tr>
<tr>
<td>.</td>
<td>&quot;</td>
</tr>
<tr>
<td>Tn</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

5.9.1 H-compact Algorithm for horizontal compaction

Let \( f \) = file used for mining

Let compact \([ \_ ][ \_]\) = compact array for horizontal data

//100000*32 for T40110D100K

while(f) // Loop till the file has records
{
   
   i=0;

   Read one record each from file
   {
      
      let c= one value from record // indicates the itemset in the transaction
   }
5.10 Vertical Compaction

The format of vertically compact data is like mentioned below:

Table 5.2 Initial data format converted to Data Format in vertical compaction

<table>
<thead>
<tr>
<th>Trans Item</th>
<th>I1</th>
<th>I2</th>
<th>...</th>
<th>...</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ItemNo #</th>
<th>Sequence of integers representing transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>Integer 1:- 32 bits=32 records</td>
</tr>
<tr>
<td></td>
<td>Integer 2:- 32 bits=32 records</td>
</tr>
<tr>
<td></td>
<td>Integer n:- 32 bits=32 records</td>
</tr>
<tr>
<td>I2</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;</td>
</tr>
<tr>
<td>In</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;</td>
</tr>
</tbody>
</table>
To convert the dataset into vertically compact data format, following algorithm was developed.

5.10.1 V-compact Algorithm for vertical compaction

Let $f$ be file used for mining.

Let $\text{compactv}[[]]$ = compact array for vertical data //1000*3125 for T40I10D100K

$k=0$;

$i=0$;

Repeat while file $f$ has records

{j=0;

while (there is record in file & & $i<1000$)

Read (each record);
For each character token in record
{
    compactv[c][j] = compactv[c][j] | M[k]; // to form a vertical compact database
}

k++;
if (k > 31)
{
    k = 0; j++;
}

i++;
}

return compactv;

Figure 5.5 Compact Dataset

64
5.11 Proposed Algorithm: FIM_GPUComp

Let:

D- Input data of transactions
C- Compact data set
R= row of compact matrix
I – Item sets
F- Frequent Item sets
Sup- Minimum Support
R=Row of frequent itemset

Output:

A- All Frequent Itemsets

Algorithm

Let A=Ø, F= Ø, C= Ø; // set the initial values to NULL
C= V-compact (D); // Convert the dataset into vertical compact format
Transfer C to GPU; //copies compact data to GPU memory
For all R ∈ C

{ 
    ni = count of number of 1s for each I // For counting support of each itemset i1,i2…..in 
    for all ni 
    {
        if (ni<sup)
            Prune(ni); // filter out the infrequent itemsets 
        else
            F=F U ni; // Append the itemset to frequent itemset list 
    }
}

Candidate_gen (F); // Generate candidates for next level
do
{
For all $I_i \in F$
{
    $f_i = R_i \text{ AND } R_{i+1}$  // Count the support of 2..3 itemsets
}
$n_i =$ count of 1s for all $f_i$
for all $n_i$
{
    if ($n_i <$ sup)
        Prune($n_i$);
    else
        $F = F \cup n_i$;  // Append the itemset to frequent itemset list
}
Candidate_gen ($F$);
}
while ($F \neq \emptyset$);
}

5.12 Support Count using GPU

Vertical data format is copied to GPU and the number of ON bits are counted by GPU. Multiple threads of GPU blocks work concurrently to count the itemsets. In first iteration, each record gives the total count of 1-itemset. Frequent itemsets are again sent to CPU for candidate generation. For finding co-occurrence of two or more itemsets, bitwise AND operation is performed and then the resultant ON bits are sent as output.
5.13 Experimental Result and Analysis

The size of dataset was ranging from 1 lac to 50 lacs the frequent itemsets were counted for up to three itemsets. Following table indicates the run time of frequent itemset mining through (i) CPU after vertically compacting data (ii) CPU+GPU after vertically compacting data.

<table>
<thead>
<tr>
<th>No. of Records</th>
<th>FIM- CPU before compaction (Secs.)</th>
<th>FIM- GPU after compaction (Secs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>25.22</td>
<td>6.36</td>
</tr>
<tr>
<td>200000</td>
<td>56.51</td>
<td>11.84</td>
</tr>
<tr>
<td>500000</td>
<td>105.02</td>
<td>18.08</td>
</tr>
<tr>
<td>1000000</td>
<td>186.04</td>
<td>28.44</td>
</tr>
<tr>
<td>5000000</td>
<td>652.82</td>
<td>88.51</td>
</tr>
</tbody>
</table>

Figure 5.6 CPU vs. GPU Execution Time
Multi fold increase in performance of FIM algorithm was observed when run using GPU. As the number of records grows, there was a notable increase of up to 3X to 7X using GPU. But when talking about CPU versions of before and after compaction, the gain in performance was very less i.e. from 1.5X to 1.9X. Space reduces after compaction, but since the data was sparse, comparatively less performance improvement was observed for CPU and CPU compact versions.

Table 5.4 Performance gain CPU vs. GPU

<table>
<thead>
<tr>
<th>Input records</th>
<th>Performance Gain: FIM-CPU vs. FIM GPU Compact</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>3.96540881</td>
</tr>
<tr>
<td>200000</td>
<td>4.77280405</td>
</tr>
<tr>
<td>500000</td>
<td>5.80862832</td>
</tr>
<tr>
<td>1000000</td>
<td>6.54149086</td>
</tr>
<tr>
<td>5000000</td>
<td>7.37566377</td>
</tr>
</tbody>
</table>

Figure 5.7 Comparison of Speed of FIM
5.14 Summary

GPU is a cost effective platform for developing parallel applications. Hardware level parallelization of GPU is exploited in this work to optimize the performance of FIM algorithm. The performance of frequent item set mining can be further improved if the input data is first converted into vertically compact form. The chapter describes the input data format, method of data compaction and a proposed parallel FIM_GPUComp algorithm. Extensive experimentation was conducted and the results indicate how the combination of data compaction technique, vertical data format and parallel execution results in significant performance improvement.